Chap. 3

ARM Assembly Language Programming

3.1 Data processing instructions

- Rules which apply to ARM data processing instructions
  - All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself
  - The result, if there is one, is 32 bits wide and is placed in a register
    - There is an exception here: long multiply instructions produce a 64-bit result (Section 5.8)
  - Each of the operand registers and the result registers are independently specified in the instruction
    - ARM uses a 3-address format for these instructions
    - In writing the assembly language source code, the operands must be written in the correct order, which is result register first, then the first operand and lastly the second operand.

Data Processing Instructions

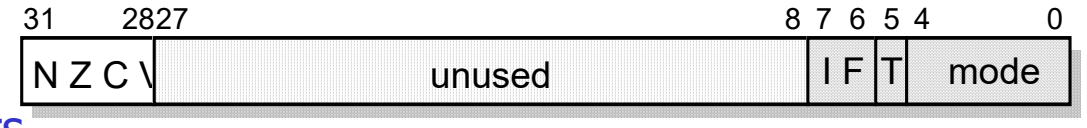
- All sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results - just set condition codes)
  - Logical operations
  - Data movement between registers
- ARM is a load / store architecture
  - These instructions only work on registers and ***NOT*** on memory.
- Perform a specific operation on one or two operands.
  - First operand always a register – Rn.
  - Second operand sent to the ALU via barrel shifter.

3.1 Data processing instructions

- Simple register operands
  - When instruction is executed, the N, Z, C and V flags in the CPSR may change
  - Arithmetic operations
    - These instructions perform binary arithmetic on two 32-bit operands

```
        ADD     r0, r1, r2          ; r0 := r1 + r2
        ADC     r0, r1, r2          ; r0 := r1 + r2 + C
        SUB     r0, r1, r2          ; r0 := r1 - r2
        SBC     r0, r1, r2          ; r0 := r1 - r2 + C - 1
Reverse subtract  RSB     r0, r1, r2          ; r0 := r2 - r1
        RSC     r0, r1, r2          ; r0 := r2 - r1 + C - 1
```

    - C flag is set to
      1, if no borrow occurs
      0, if a borrow does occur



3.1 Data processing instructions

- Simple register operands
  - Bit-wise logical operations
    - These instructions perform the specified Boolean logic operation on each bit pair of the input operands
    - AND: r0[*i*] := r1[*i*] AND r2[*i*] for each value of *i* from 0 to 31
    - BIC: 'bit clear' where every '1' in the second operand clears the corresponding bit in the first

```
AND     r0, r1, r2          ; r0 := r1 and r2
ORR     r0, r1, r2          ; r0 := r1 or r2
EOR     r0, r1, r2          ; r0 := r1 xor r2
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

3.1 Data processing instructions

- Simple register operands
  - Register movement operations
    - These instructions ignore the first operand, which is omitted from the assembly language format, and simply move the second operand to the destination

```
MOV     r0, r2          ; r0 := r2
MVN     r0, r2          ; r0 := not r2
```

  - Comparison operations
    - These instructions do not produce a result but just set the condition code bits (N, Z, C and V) in the CPSR according to the selected operation

compare	CMP	r1, r2	; set cc on r1 - r2
compare negated	CMN	r1, r2	; set cc on r1 + r2
test	TST	r1, r2	; set cc on r1 and r2
test equal	TEQ	r1, r2	; set cc on r1 xor r2

3.1 Data processing instructions

- Immediate operands
  - If, instead of adding two registers, we simply wish to add a constant to a register we can replace the second source operand with an immediate value, which is a literal constant, preceded by '#':

```
ADD    r3,  r3,  #1          ; r3 := r3 + 1
AND    r8,  r7,  #&ff        ; r8 := r7 AND ff16 := r7 [7:0]
```

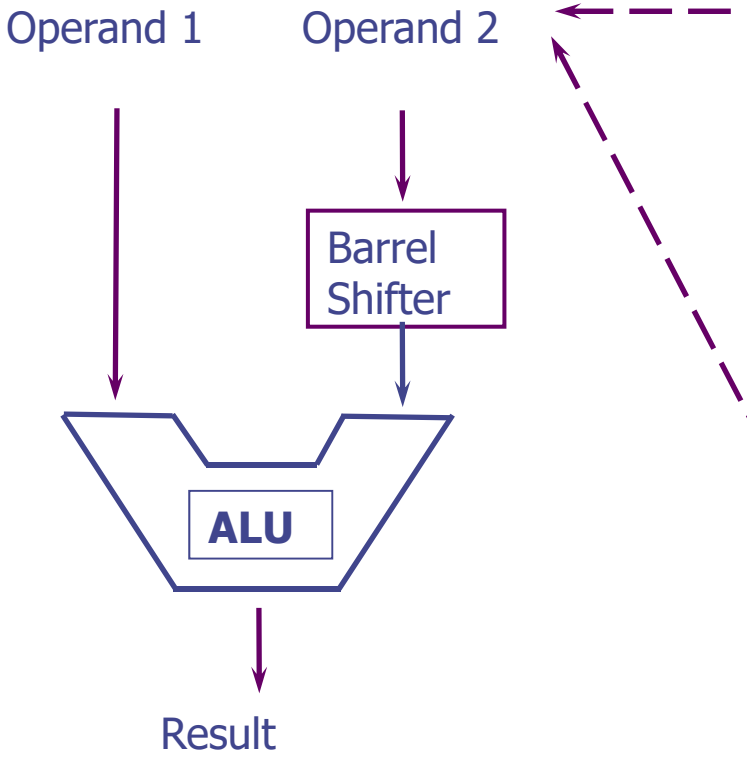
  - The second example shows that the immediate value may be specified in hexadecimal (base 16) notation by putting '&' after the '#' (***Note: #&ff is equivalent to #0xff*** in gcc ARM assembly)
  - Most valid immediate values are given by

$$immediate = (0 \rightarrow 255) \times 2^{2n}, \quad \text{where } 0 \leq n \leq 12$$

    - the assembler will also replace MOV with MVN, ADD with SUB, and so on, where this can bring the immediate within range
    - Any number which can be represented as (0->255) ror (0,2,…30)

The Barrel Shifter

- ARM has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.



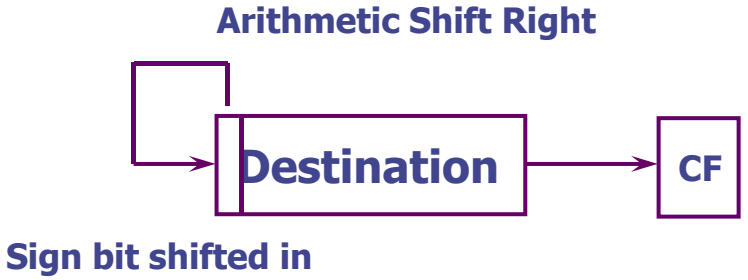
- **Register, optionally with shift operation applied.**
- **Shift value can be either be:**
  - 5 bit unsigned integer
  - specified in bottom byte of another register.
- **Immediate value**
  - 8 bit number
  - can be rotated right through an even number of positions.(0,2,4,…,30)
  - assembler will calculate rotate for you from constant.

Shift and Rotation Operations

- Shifts left by the specified amount (multiplies by powers of two) e.g.
  - LSL #5 = multiply by 32
  - LSL = ASL

- Logical Shift Right
  - LSR #5 = divide by 32 (unsigned)

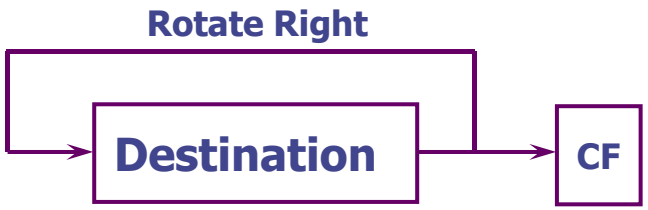
- Arithmetic Shift Right and preserves the sign bit, for 2's complement operations.
  - ASR #5 = divide by 32 (signed)



Ref. [4]

Barrel Shifter - Rotations

- Rotate Right (ROR)
  - Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB. E.g. ROR #5.
  - Note the last bit rotated is also used as the Carry Out.



- Rotate Right Extended (RRX):

  uses the CPSR C flag as a

  33rd bit.
  - Rotates right by 1 bit (only)
  - Encoded as ROR #0

Ref. [4]

Second Operand: Shifted Register

- The amount by which the register is to be shifted is contained in either:
  - the immediate 5-bit field in the instruction
    - NO OVERHEAD, shift is done for free - executes in single cycle.

      ADD r5, r5, r3, LSL #3;
  - the *bottom byte* of a register (not PC)
    - This then takes extra cycle to execute
    - ARM doesn't have enough read ports to read 3 registers at once.

      ADD r5, r5, r3, LSL r2;

- If no shift is specified then a default shift **LSL #0** is applied
  - i.e. barrel shifter has no effect on value in register.

Second Operand: Shifted Register *(cont'd)*

- Using a multiplication instruction to multiply by a constant
  - load the constant into a register
  - wait for a number of internal cycles for the multiplication to complete
- A more optimum solution can be often found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
  - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.
- Example:

```
r0   = r1 * 5
     = r1 + (r1 * 4)

     ADD r0, r1, r1, LSL #2
```

- Example:

```
r2 = r3 * 105
   = r3 * 15 * 7
   = r3 * (16 - 1) * (8 - 1)
     RSB r2, r3, r3, LSL #4    ; r2 = r3 * 15
     RSB r2, r2, r2, LSL #3    ; r2 = r2 * 7
```

Second Operand: Immediate Value

- There is no single instruction that will load a 32 bit immediate constant into a register without performing a data load from memory.
- The data processing instruction format has 12 bits available for operand2
  - If used directly this would only give a range of 4096.
- Instead, it is used to store 8-bit constants, giving a range of 0 - 255.
- These 8 bits can then be rotated right through an even number of positions (i.e. RORs by 8, 10, 12, …, 30).
  - $(0 \rightarrow 255) \times 2^{2n}$ (n=0, 1, 2, …, 12)
  - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory.

TABLE 6.1
Examples of Creating Constants with Rotation

Rotate	Binary	Decimal	Step	Hexadecimal
No rotate	00000000000000000000000xxxxxxxx	0-255	1	0-0xFF
Right, 30 bits	00000000000000000000xxxxxxxx00	0-1020	4	0-0x3FC
Right, 28 bits	0000000000000000000xxxxxxxx0000	0-4080	16	0-0xFF0
Right, 26 bits	000000000000000000xxxxxxxx000000	0-16320	64	0-0x3FC0
…	…	…	…	…
Right, 8 bits	xxxxxxxx000000000000000000000000	$0\text{-}255x2^{24}$	2^{24}	0-0xFF000000
Right, 6 bits	xxxxxx000000000000000000000000xx	—	—	—
Right, 4 bits	xxxx000000000000000000000000xxxx	—	—	—
Right, 2 bits	xx000000000000000000000000000000	—	—	—

Second Operand: Immediate Value *(cont'd)*

- This gives us:
  - 0 - 255 [0 - 0xff]
  - 256,260,264,..,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
  - 1024,1040,1056,..,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
  - 4096,4160, 4224,..,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]
- These can be loaded using, for example:
  - MOV r0, #0x40 ror 26 ; => MOV r0, #0x1000 (ie 4096)
- To make this easier, the assembler will convert to this form for us if simply given the required constant:
  - MOV r0, #4096 ; => MOV r0, #0x1000 (ie 0x40 ror 26)
- The bitwise complements can also be formed using MVN:
  - MOV r0, #0xFFFFFFFF ; assembles to MVN r0, #0
- If the required constant cannot be generated, an error will be reported.

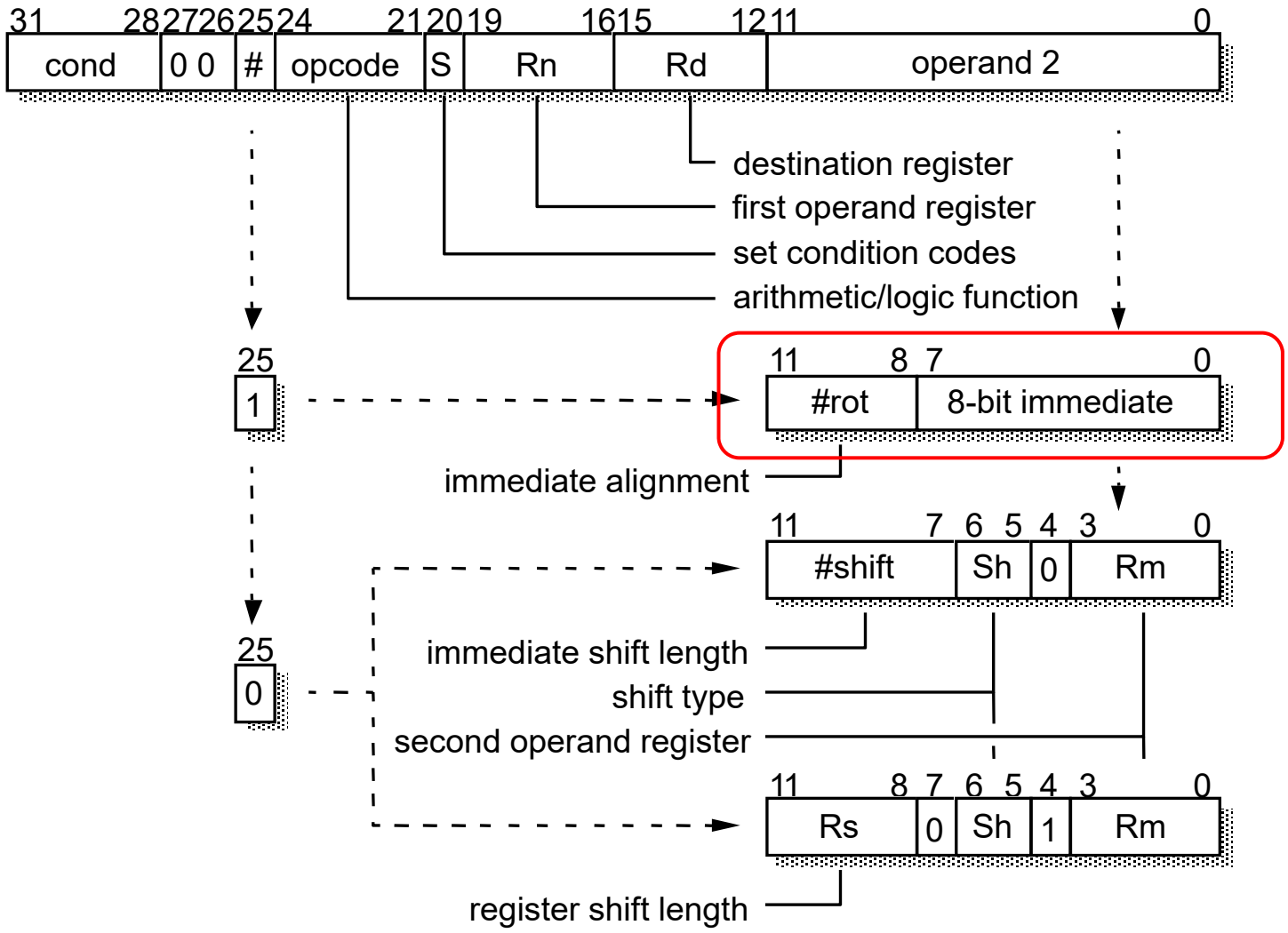


Fig. 5.6 Data processing instruction binary encoding

MOV instruction

- MOV Rd, Rm
- Condition flags: If S is specified, the instruction will
  - Update the N and Z flags according to the result
  - Do not affect the V flag
  - The carry flag is updated to the last bit shifted out of Rm.

Examples

```
ADD     r3,r7,#1020          ; immed_8r. 1020 is 0xFF rotated right by 30 bits.
AND     r0,r5,r2             ; r2 contains the data for Operand2.
SUB     r11,r12,r3,ASR #5    ; Operand2 is the contents of r3 divided by 32.
MOVS    r4,r4, LSR #32       ; Updates the C flag to r4 bit 31. Clears r4 to 0.
```

Incorrect examples

```
ADD     r3,r7,#1023          ; 1023 (0x3FF) is not a rotated 8-bit pattern.
SUB     r11,r12,r3,LSL #32   ; #32 is out of range for LSL.
MOVS    r4,r4,RRX #3         ; Do not specify a shift amount for RRX. RRX is
                             ; always a one-bit shift.
```

Example

- PRE
  - cpsr = nzcvqiFt_USER
  - r0=0x00000000
  - r1=0x80000004
- MOVS r0, r1, LSL #1
- POST
  - cpsr = nzCvqiFt_USER
  - r0=0x00000008
  - r1=0x80000004

3.1 Data processing instructions

- Setting the condition codes (N, Z, C and V)
  - The comparison operations only set the condition codes, so there is no option with them
  - For all other data processing instructions a specific request must be made
  - At the assembly language level this request is indicated by adding an 'S' to the opcode, standing for 'Set condition codes'

```
ADDS    r2,  r2,  r0 ;  32-bit carry out -> C . .
ADC     r3,  r3,  r1 ;  . . and added into high word
```

    - The code performs a 64-bit addition of two number held in r0-r1 and r2-r3

- Use of the condition codes
  - Control the program flow through the conditional branch instructions (Section 3.3)

Condition Flags

	Logical Instruction	Arithmetic Instruction
Flag		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

3.1 Data processing instructions

- Multiplies
  - Multiplication
    - MUL r4, r3, r2 ; r4 := $(r3 \times r2)_{[31:0]}$
  - There are some important differences from the other arithmetic instructions:
    - Immediate second operands are not supported
    - The result register must not be the same as the first source register
    - If the 'S' bit is set the V flag is preserved (as for a logical instruction) and the C flag is rendered meaningless
  - Multiplying two 32-bit integers gives a 64-bit result, the least significant 32 bits of which are placed in the result register and the rest are ignored
    - ARM also support long multiply instructions which place the most significant 32 bits into a second result register (Section 5.8)

3.1 Data processing instructions

- Multiplies (Cont.)
  - Multiply-accumulate instruction

```
MLA     r4,   r3,   r2,   r1      ; r4 := (r3 × r2 + r1)[31:0]
```

  - Multiplication by a constant
    - can be implemented by loading the constant into a register and then using one of these instructions, but it is usually more efficient to use a short series of data processing instructions using shifts and adds or subtracts

```
ADD     r0,   r0,   r0,   LSL  #2      ; r0' := 5 × r0
RSB     r0,   r0,   r0,   LSL  #3      ; r0'' := 7 × r0' (= 35 × r0)
```

Example

- PRE
  - r0=0x00000000
  - r1=0x00000002
  - r2=0x00000002
- MUL r0, r1, r2
- POST
  - r0=0x00000004
  - r1=0x00000002
  - r2=0x00000002

- PRE
  - r0=0x00000000
  - r1=0x00000000
  - r2=0xf0000002
  - r3=0x00000002
- UMULL r0, r1, r2,r3
- POST
  - r0=0xe0000004
  - r1=0x00000001

3.2 Data transfer instructions

- Three basic forms of data transfer instruction in the ARM instruction set
  - Single register load and store instructions
    - These instructions provide the most flexible way to transfer single data items between an ARM register and memory
  - Multiple register load and store instructions
    - These instructions are less flexible than single register transfer instructions, but enable large quantities of data to be transferred more efficiently
  - Single register swap instructions
    - These instructions allow a value in a register to be exchanged with a value in memory, effectively doing both a load and a store operation in one instruction

3.2 Data transfer instructions

- Register-indirect addressing
  - The ARM data transfer instructions are all based around register-indirect addressing, with modes that include base-plus-offset and base-plus-index addressing
  - Register-indirect addressing uses a value in one register (the base register) as a memory address and either loads the value from that address into another register or stores the value from another register into that memory address

- Single register load and store instructions
  - The notation used here indicates that the data quantity is the 32-bit memory word addressed by r1

```
LDR     r0,  [r1]       ; r0  :=  mem32[r1]
STR     r0,  [r1]       ; mem32 [r1]  :=  r0
```

3.2 Data transfer instructions

- Single register load and store instructions
  - These instructions compute an address for the transfer using a base register, which should contain an address near to the target address, and an offset which may be another register (base-plus-index) or an immediate value (base-plus-offset)
  - An unsigned 12bit immediate value (i.e. 0 - 4095 bytes).

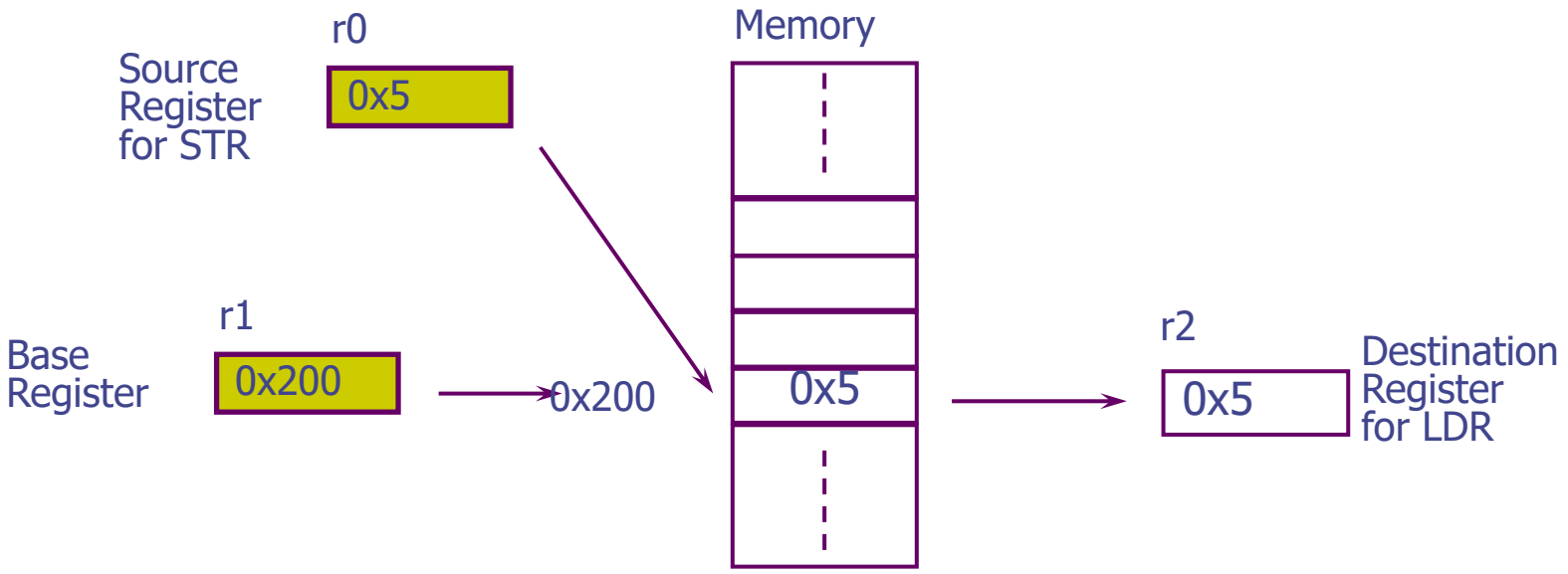
```
LDR    r0,   [ r1 , #4 ]        ;  r0  :=  mem32 [ r1 + 4 ]
```

  - A register, optionally shifted by an immediate value

```
STR    r0,   [ r1 , r2, LSL #2 ]        ; mem32 [ r1 + 4*r2 ] := r0
```

Load and Store Word or Byte: Base Register

- The memory location to be accessed is held in a base register
  - STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.
  - LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.



Ref. [8]

3.2 Data transfer instructions

- Initializing an address pointer
  - To load or store from or to a particular memory location, an ARM register must be initialized to contain the address of the location
  - A data processing instruction can be employed to add a small offset to r15, but calculating the appropriate offset may not be that straightforward
  - ARM assemblers have an inbuilt 'pseudo instruction', ADR, which makes this easy

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

An example: copy words from one table to the other

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1]            ; get TABLE1 1st word
        STR     r0, [r2]            ; copy into TABLE2
        ADD     r1, r1, #4          ; step r1 on 1 word
        ADD     r2, r2, #4          ; step r2 on 1 word
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

3.2 Data transfer instructions

- Base plus offset addressing
  - If the base register does not contain exactly the right address, an offset may be added (or subtracted) to the base to compute the transfer address:

    LDR r0, [r1 , #4] ; r0 := mem_{32} [r1 + 4]

    This is a pre-indexed addressing mode

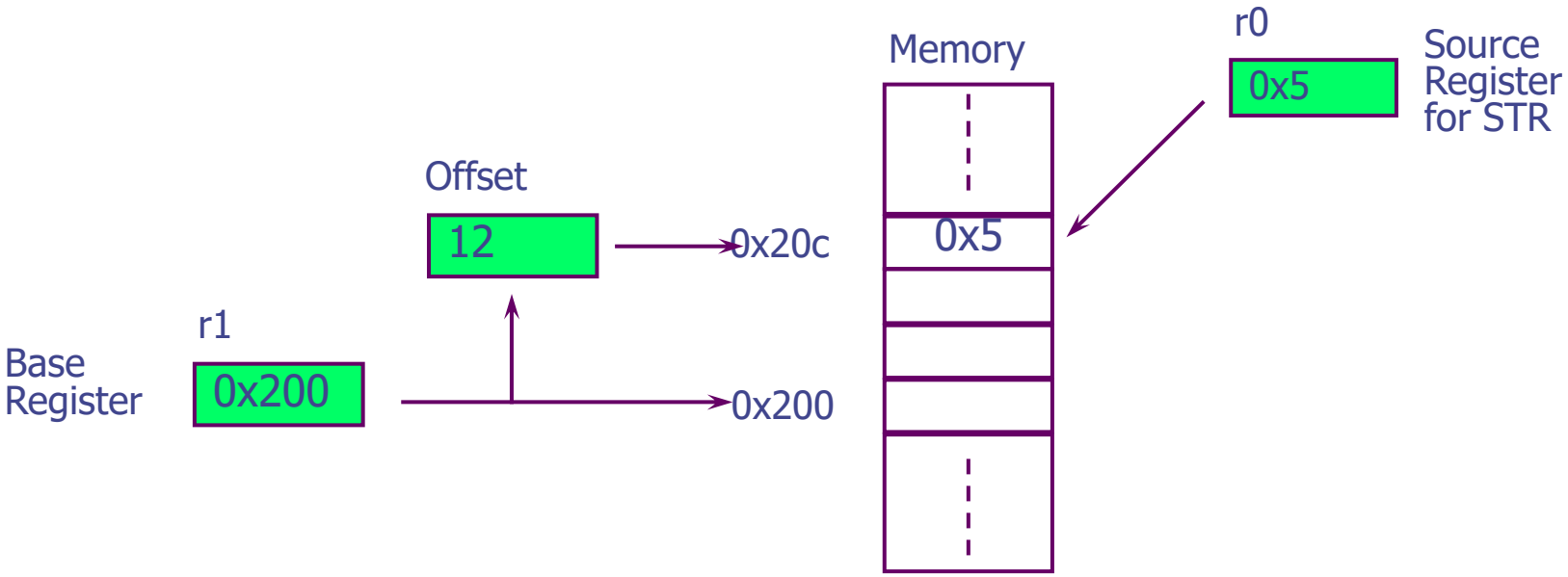
  - Pre-indexed addressing with auto-indexing

    LDR r0, [r1 , #4] ! ; r0 := mem_{32} [r1 + 4]
    ; r1 := r1 + 4

  - On the ARM this auto-indexing costs no extra time since it is performed on the processor's datapath while the data is being fetched from memory

Load and Store: Pre-indexed Addressing

- Example: **STR r0, [r1,#12]**

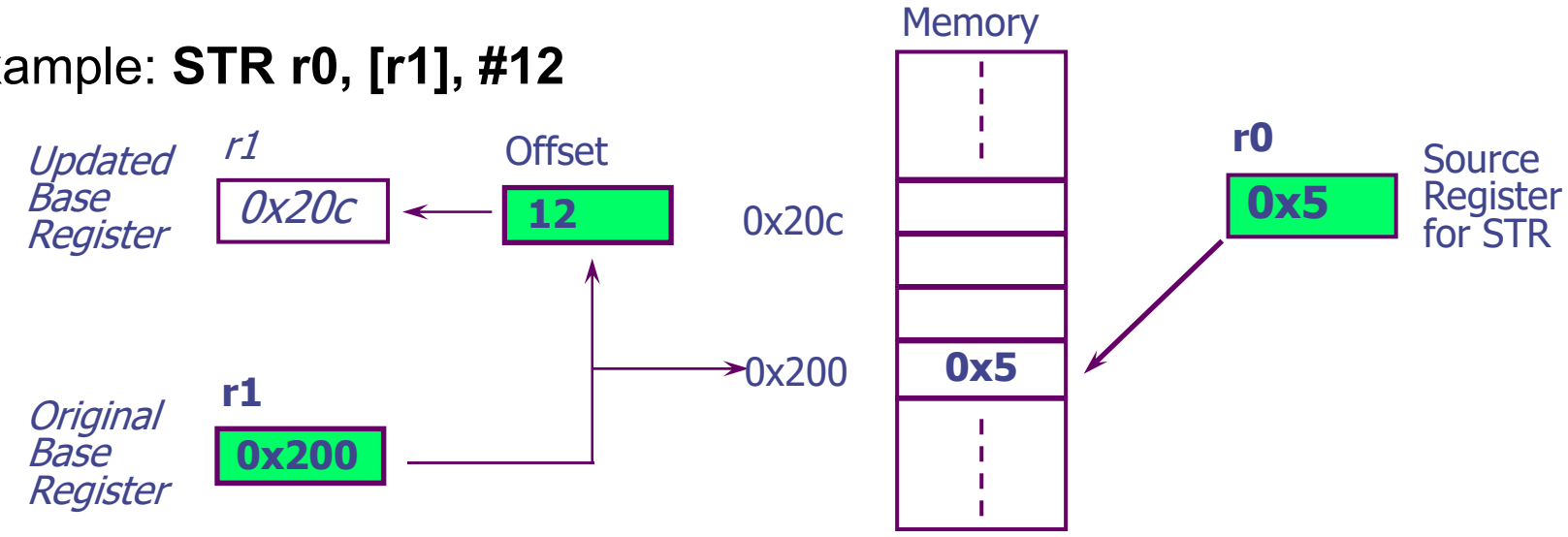


- To store to location 0x1f4: **STR r0, [r1,# -12]**
- To auto-increment base pointer to 0x20c: **STR r0, [r1, #12]!**
- If r2 contains 3, access 0x20c by multiplying this by 4:
  - STR r0, [r1, r2, LSL #2]

Ref. [8]

Load and Store: Post-indexed Addressing

- Example: **STR r0, [r1], #12**



- To auto-increment the base register to location 0x1f4:
  - STR r0, [r1], #-12
- If r2 contains 3, auto-incremenet base register to 0x20c by multiplying this by 4:
  - STR r0, [r1], r2, LSL #2

3.2 Data transfer instructions

- Base plus offset addressing (Cont.)
  - Post-indexed addressing

```
LDR     r0, [r1], #4      ; r0 := mem32[r1]
                          ; r1 := r1 + 4
```

  - For example

```
COPY    ADR     r1, TABLE1        ; r1 points to TABLE1
        ADR     r2, TABLE2        ; r2 points to TABLE2
LOOP    LDR     r0, [r1], #4      ; get TABLE1 1st word
        STR     r0, [r2], #4      ; copy into TABLE2
        ???                       ; if more go back to LOOP
        ..
TABLE1                            ; < source of data >
        ..
TABLE2                            ; < destination >
        ..
```

Loading Addresses into registers

```
SRAM_BASE     EQU    0x04000000
              AREA   EXAMPLE, CODE, READONLY
;
; initialization section
;
              ENTRY
              MOV    r0, #SRAM_BASE
              MOV    r1, #0xFF000000
              .
              .
              .
```

- What if #SRAM_BASE is changed to an address value which can not be simply moved to register?

Implementation of LDR rx, =0x...

```
            AREA Example, CODE
            ENTRY                       ; mark first instruction
            BL      func1               ; call first subroutine
            BL      func2               ; call second subroutine
stop        B       stop                ; terminate the program
func1       LDR     r0, =42             ; => MOV r0, #42
            LDR     r1, =0x12345678     ; => LDR r1, [PC, #N]
                                        ; where N=offset to literal pool 1
            LDR     r2, =0xFFFFFFFF     ; => MVN r2, #0
            BX      lr                  ; return from subroutine
            LTORG                       ; literal pool 1 has 0x12345678
func2       LDR     r3, =0x12345678     ; => LDR r3, [PC, #N]
                                        ; N=offset back to literal pool 1
            ;LDR    r4, =0x87654321     ; if this is uncommented, it fails.
                                        ; Literal pool 2 is out of reach!
            BX      lr                  ; return from subroutine
BigTable
            SPACE 4200                  ; clears 4200 bytes of memory,
                                        ; starting here
            END                         ; literal pool 2 empty
```

```
            AREA Example, CODE
            ENTRY                       ; mark first instruction
            BL      func1               ; call first subroutine
            BL      func2               ; call second subroutine
stop        B       stop                ; terminate the program
func1       LDR     r0, =42             ; => MOV r0, #42
            LDR     r1, =0x12345678     ; => LDR r1, [PC, #N]
                                        ; where N=offset to literal pool 1
            LDR     r2, =0xFFFFFFFF     ; => MVN r2, #0
            BX      lr                  ; return from subroutine
            LTORG                       ; literal pool 1 has 0x12345678
func2       LDR     r3, =0x12345678     ; => LDR r3, [PC, #N]
                                        ; N=offset back to literal pool 1
            ;LDR    r4, =0x87654321     ; if this is uncommented, it fails.
                                        ; Literal pool 2 is out of reach!
            BX      lr                  ; return from subroutine
BigTable
            SPACE 4200                  ; clears 4200 bytes of memory,
                                        ; starting here
            END                         ; literal pool 2 empty
```

Address		Instruction			
0x00000000	EB000001	BL	0x0000000C		
0x00000004	EB000005	BL	0x00000020		
0x00000008	EAFFFFFE	B	0x00000008		
0x0000000C	E3A0002A	MOV	RO,#0x0000002A		
0x00000010	E59F1004	LDR	R1,[PC,#0x0004]		EXECUTE
0x00000014	E3E02000	MVN	R2,#0x00000000		DECODE
0x00000018	E12FFF1E	BX	R14	← PC	FETCH
0x0000001C	12345678			← PC + 4	
0x00000020	E51F300C	LDR	R3,[PC,#-0x000C]		
0x00000024	E12FFF1E	BX	R14		

FIGURE 6.6 Disassembly of ARM7TDMI program.

Load of address into register (LDR may fail)

```
        AREA    LDRlabel, CODE, READONLY
        ENTRY                       ; Mark first instruction to execute
start
        BL      func1               ; branch to first subroutine
        BL      func2               ; branch to second subroutine
stop    B       stop                ; terminate

func1
        LDR     r0, =start          ;=> LDR R0, [PC, #offset into Literal Pool 1]
        LDR     r1, =Darea+12       ;=> LDR R1, [PC, #offset into Lit. Pool 1]
        LDR     r2, =Darea+6000     ;=> LDR R2, [PC, #offset into Lit. Pool 1]
        BX      lr                  ; return

        LTORG
func2
        LDR     r3, =Darea+6000     ; => LDR R3, [PC, #offset into Lit. Pool 1]
                                    ; (sharing with previous literal)
        ; LDR   r4, =Darea+6004    ; if uncommented produces an error
                                    ; as literal pool 2 is out of range
        BX      lr                  ; return
Darea
        SPACE   8000                ; starting at the current location, clears
                                    ; an 8000-byte area of memory to zero
        END                         ; literal pool 2 is out of range of the LDR
                                    ; instructions above
```

Loading Addresses/Symbols into registers

```
SRAM_BASE  EQU     0x04000000
           AREA    FILTER, CODE

dest       RN0     ; destination pointer
image      RN1     ; image data pointer
coeff      RN2     ; coefficient table pointer
pointer    RN3     ; temporary pointer
           ENTRY
           CODE32
Main
           ; initialization area
           LDR     dest, =#SRAM_BASE    ; move memory base into dest
           MOV     pointer, dest        ; current pointer is destination
           ADR     image, image_data    ; load image data pointer
           ADR     coeff, cosines       ; load coefficient pointer
           BL      filter               ; execute one pass of filter
           .
           .
           .
           ALIGN
image_data
           DCW     0x0001,0x0002,0x0003,0x0004
           DCW     0x0005,0x0006,0x0007,0x0008
           .
           .
           .
cosines
           DCW     0x3ec5,0x3537,0x238e,0x0c7c
           DCW     0xf384,0xdc72,0xcac9,0xc13b
           .
           .
           .
           END
```

Pseudo instructions: ADR and LDR

Address		*instruction*	
?		ADR r1, TEXT	=> *ADD r1, r15,#?*
?	LOOP	LDRB r0, [r1]	
?		CMP r0, #0	
?		SWINE #0	
?		BNE LOOP	
?		SWINE #11	
?	TEX	= "NSYSU"	

	instruction
	LDR r2, =0x87654321
	ADR r1, TEX
LOOP	LDRB r0, [r1]
	CMP r0, #0
	SWINE #0
	BNE LOOP
	SWINE #11
TEX	= "NSYSU"

Address	
720	enc(LDR, r2, [r15, #28])
724	enc(ADD r1, r15, #28)
728	enc(LDRB r0, [r1])
732	enc(CMP r0, #0)
736	enc(SWINE #0)
740	enc(BNE LOOP)
744	enc(SWINE #11)
748	0x87654321
752	"NSYS"
756	"U"

Use of ADR may fail

```
        AREA    adrlabel,CODE,READONLY
        ENTRY                           ; mark first instruction to execute

Start   BL      func                    ; branch to subroutine
stop    B       stop                    ; terminate
        LTORG                           ; create a literal pool
func    ADR     r0, Start               ; => SUB r0, PC, #offset to Start
        ADR     r1, DataArea            ; => ADD r1, PC, #offset to DataArea
        ;ADR    r2, DataArea+4300       ; This would fail because the offset
                                        ; cannot be expressed by operand2 of ADD
        ADRL    r2, DataArea+4300       ; => ADD r2, PC, #offset1
                                        ; ADD r2, r2, #offset2
        BX      lr                      ; return
DataArea
        SPACE   8000                    ; starting at the current location,
                                        ; clears an 8000-byte area of memory to 0
        END
```

3.2 Data transfer instructions

- Base plus offset addressing (Cont.)
  - The size of the f\data item which is transferred is a single unsigned 8-bit byte instead of a 32-bit word

```
LDRB    r0,  [r1]          ;  r0  :=  mem8 [r1]
```

Example

- PRE
  - r0=0x00000000
  - r1=0x00090000
  - mem32[0x00090000]=0x01010101
  - mem32[0x00090004]=0x02020202
- LDR r0, [r1, #4]!
  - r0=0x02020202
  - r1=0x00090004
- LDR r0, [r1, #4]
  - r0=0x02020202
  - r1=0x00090000
- LDR r0, [r1], #4
  - r0=0x01010101
  - r1=0x00090004

```
LDR  r5, [r3]              ; load r5 with data from ea<r3>
STRB r0, [r9]              ; store data in r0 to ea<r9>
STR  r3, [r0, r5, LSL #3]  ; store data in r3 to ea<r0+(r5<<3)>
LDR  r1, [r0, #4]!         ; load r1 from ea<r0+4>,r0=r0+4
STRB r7, [r6, #-1]!        ; store byte to ea<r6-1>,r6=r6-1
LDR  r3, [r9], #4          ; load r3 from ea<r9>,r9=r9+4
STR  r2, [r5], #8          ; store word to ea<r5>,r5=r5+8
```

```
STR r3, [r0, r5, LSL #3]      ; store r3 to ea<r0+(r5<<3)> (r0 unchanged)
LDR r6, [r0, r1, ROR #6]!     ; load r6 from ea<r0+(r1>>6)> (r0 updated)
LDR r0, [r1, #-8]             ; load r0 from ea<r1-8>
LDR r0, [r1, -r2, LSL #2]     ; load r0 from ea<r1+(-r2<<2)>
LDRSH r5, [r9]                ; load signed halfword from ea<r9>
LDRSB r3, [r8, #3]            ; load signed byte from ea<r8+3>
LDRSB r4, [r10, #0xc1]        ; load signed byte from ea<r10+193>
```

```
SRAM_BASE   EQU     0x04000000      ; start of SRAM for STR910FM32
            AREA    StrCopy, CODE
            ENTRY                   ; mark the first instruction
Main        ADR     r1, srcstr      ; pointer to the first string
            LDR     r0, =SRAM_BASE  ; pointer to the second string
strcopy
            LDRB    r2, [r1], #1    ; load byte, update address
            STRB    r2, [r0], #1    ; store byte, update address
            CMP     r2, #0          ; check for zero terminator
            BNE     strcopy         ; keep going if not
stop        B       stop            ; terminate the program
srcstr      DCB     "This is my (source) string", 0
            END
```

Endianness

- r1=0x0A0B0C0D r2=0x400 STR r1,[r2]

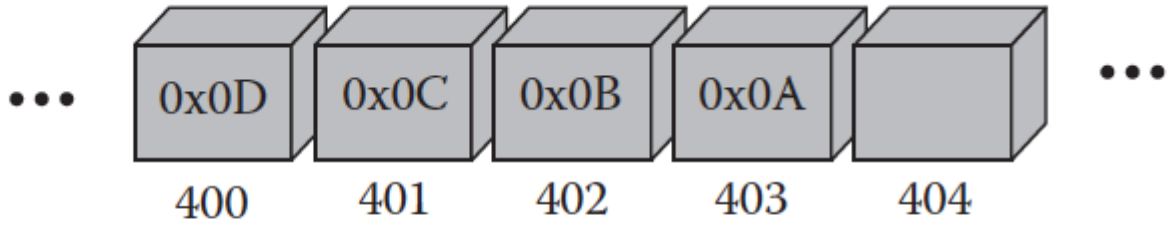


FIGURE 5.3 Little-endian memory configuration.

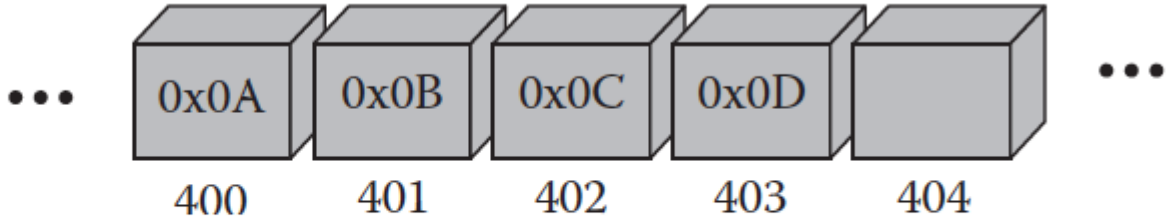


FIGURE 5.4 Big-endian memory configuration.

ARM syntax vs GNU syntax assembly

- There are two popular assemblers used which adopts slightly different ARM syntaxes.
  - ARM syntax

```
; Iterate round a loop 10 times, adding 1 to a register each time.
        AREA ||.text||, CODE, READONLY, ALIGN=2

main PROC
        MOV     w5,#0x64        ; W5 = 100
        MOV     w4,#0           ; W4 = 0
        B       test_loop       ; branch to test_loop
loop
        ADD     w5,w5,#1        ; Add 1 to W5
        ADD     w4,w4,#1        ; Add 1 to W4
test_loop
        CMP     w4,#0xa         ; if W4 < 10, branch back to loop
        BLT     loop
        ENDP

        END
```

  - GNU syntax

```
        .section .text,"x"
        .align 2

main:
        MOV     w5,#0x64        // W5 = 100
        MOV     w4,#0           // W4 = 0
        B       test_loop       // branch to test_loop
loop:
        ADD     w5,w5,#1        // Add 1 to W5
        ADD     w4,w4,#1        // Add 1 to W4
test_loop:
        CMP     w4,#0xa         // if W4 < 10, branch back to loop
        BLT     loop
        .end
```

section

- .text
  - Code, constant,
- .data
  - Initialized data
  - int var=10;
- .bss
  - Uninitialized data
  - int a, b;

ARM syntax vs GNU syntax assembly

- ARM provide two methods for specifying hexadecimal literals
  - ADD r1, #0xAF
    - GNU only supports “#0x” format
  - ADD r1, #&AF
- Data definition directives

ARM	GNU
DCB	.byte
DCW	.hword
DCD	.word
DCQ	.quad
SPACE	.space

- Definition of Strings
  - ARM symbol = “NSYSU”
  - GNU .ascii = “NSYSU”
    - .asciiz (padded with a 0x00 byte automatically)
- .align vs ALIGN

GAS (Gnu ASsembler example)

```
@this is comment@the information that tells arm-none-eabi-as what arch. to assemble to
        .cpu arm926ej-s
        .fpu softvfp
@this is code section
@note, we must have the main function for the simulator's linker script
        .text
        .align      2   @align 4 byte
        .global     main
main:
        @prologue
        stmfd       sp!, {fp, lr}
        add         fp, sp, #4
        @code body
        ldr         r0, =string1
        bl printf
        ldr         r0, =string0
        ldr r1, Label1 + 8          @why not =Label1+8
        bl printf
        mov r0, #1
        bl fun
        mov         r0, #0
        @epilogue
        sub         sp, fp, #4
        ldmfd       sp!, {fp, lr}
        bx          lr
@another function
fun:
        add r0, r0, #1
        bx lr
@data section
Label1:
        .word   0x77777777
        .short  0x1122
        .align 2
        .byte   0x31, 0x32, 0x33, 0x34
        .byte   0x00
string0:
        .asci "Hello,CodeSourcery:0x%X\n\0“
string1:
        .asciz "Hello, arm-gcc\n"
.end
```

Defining Memory Areas

- table DCB 0xFE, 0XF9, 0x12, 0x34

  DCB 0x11, 0x22, 0x33, 0x44

Address	Data Value
0x4000	0xFE
0x4001	0xF9
0x4002	0x12
0x4003	0x34
0x4004	0x11
0x4005	0x22
0x4006	0x33
0x4007	0x44

- Table DCD 0xFEF91234

  DCD 0x11223344

Address	Data Value
0x4000	0x34
0x4001	0x12
0x4002	0xF9
0x4003	0xFE
0x4004	0x44
0x4005	0x33
0x4006	0x22
0x4007	0x11

3.2 Data transfer instructions

- Multiple register data transfers
  - These instructions allow any subset (or all) of the 16 registers to be transferred with a single instruction

```
LDMIA     r1,  {r0, r2, r5}    ;  r0  :=  mem32 [r1]
                                ;  r2  :=  mem32 [r1 + 4]
                                ;  r5  :=  mem32 [r1 + 8]
```

  - Since the transferred data items are always 32-bit words, the base address (r1) should be word-aligned
  - The order of the registers within the list is insignificant and does not affect the order of transfer or the values in the registers after the instruction has executed

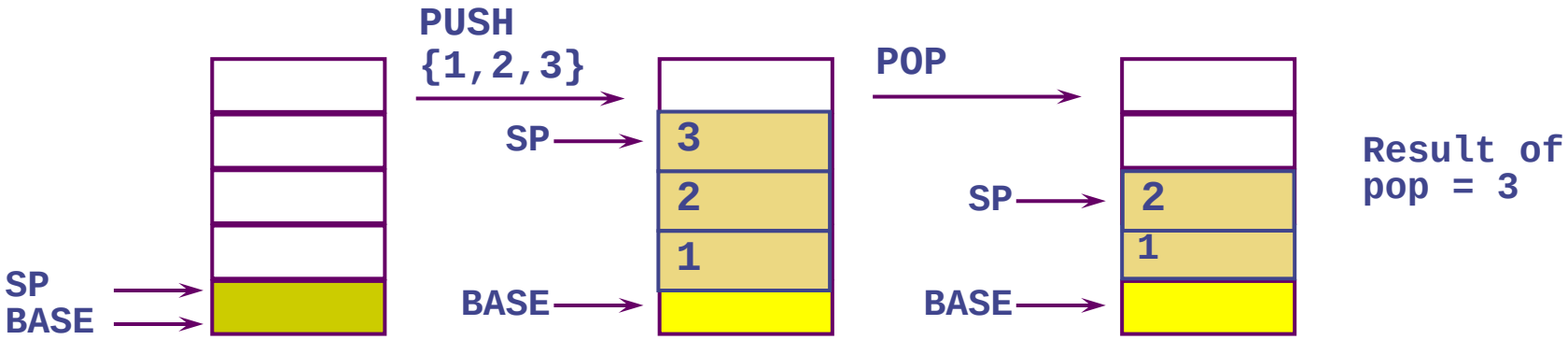
3.2 Data transfer instructions

- Stack addressing
  - A stack is usually implemented as a linear data structure which grows up (an ascending stack) or down (a descending stack) memory as data is added to it and shrinks back as data is removed
  - A stack pointer holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a full stack), or by pointing to the vacant slot where the next data item will be placed (an empty stack)
  - All four forms of stack:
    - Full ascending, Empty ascending, Full descending, Empty descending

Stacks

- A stack is an area of memory which grows as new data is “pushed” onto the “top” of it, and shrinks as data is “popped” off the top.

- Two pointers define the current limits of the stack.
  - A base pointer (frame pointer): used to point to the “bottom” of the stack (the first location).
  - A stack pointer: used to point the current “top” of the stack.



Ref. [8]

3.2 Data transfer instructions

- Block copy addressing
  - different view of multiple register transfer instruction

```
STMFD    r13!,   {r2-r9}      ; save regs onto stack
LDMIA    r0!,    {r2-r9}      ; restore from stack
STMIA    r1,     {r2-r9}
LDMFD    r13!,   {r2-r9}
```

  - r0 has increased by 32 since the '!' causes it to auto-index across eight words
  - 'FD' postfix: Full Descending stack address mode
  - 'IA' postfix: Increment After

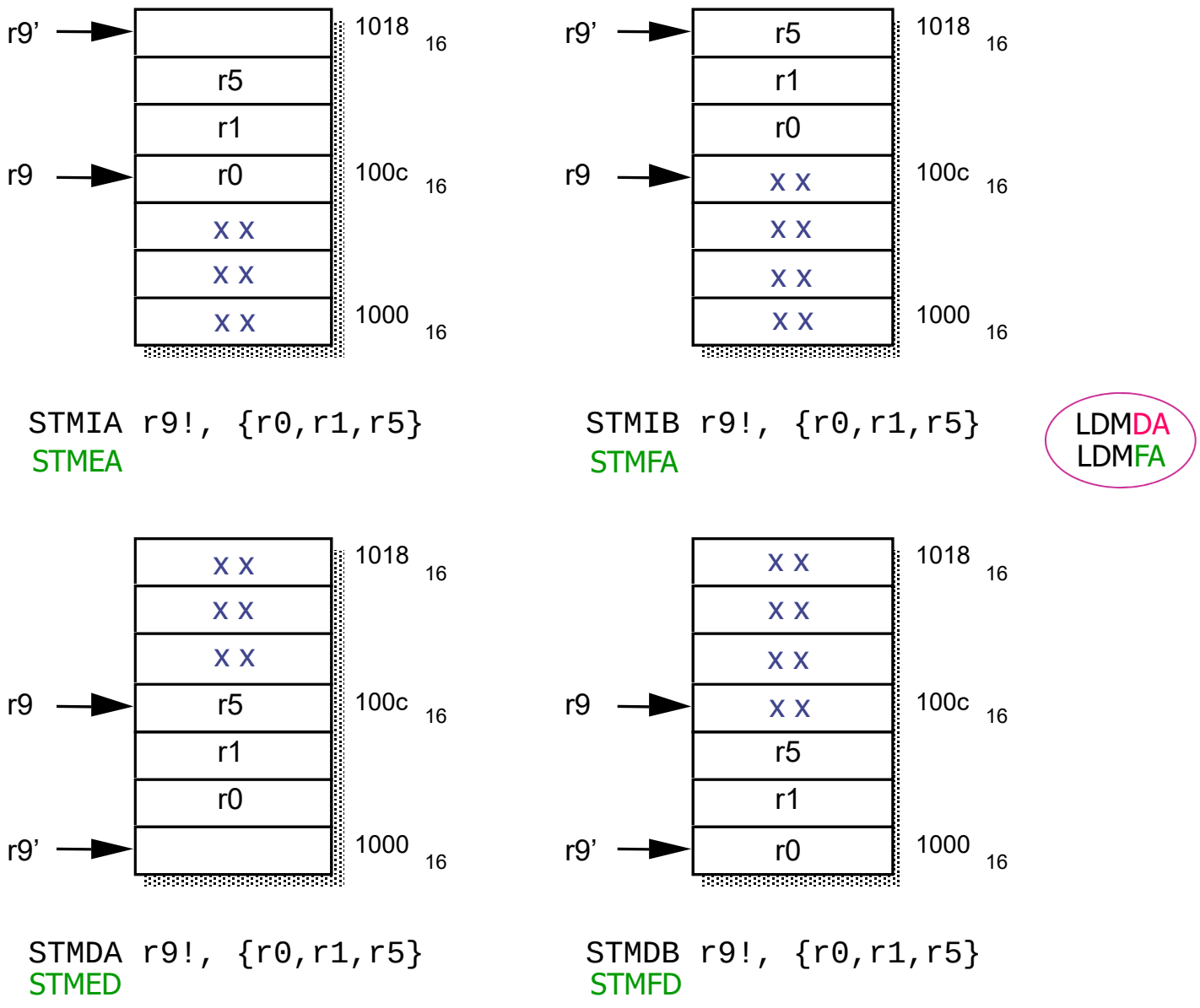


Fig. 3.2 Multiple register transfer addressing modes

LDM/STM operations

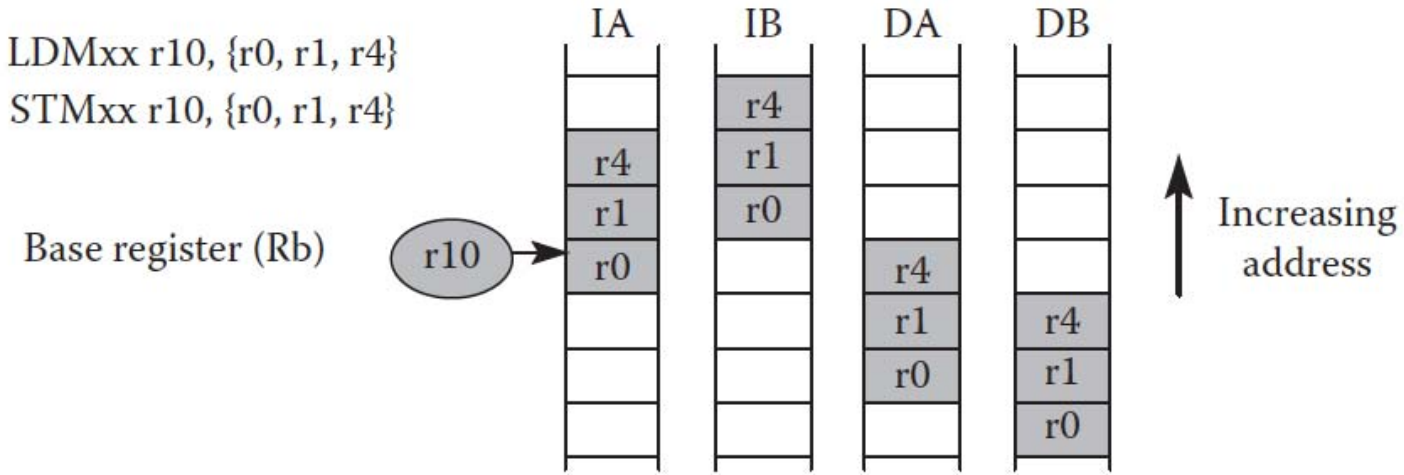


FIGURE 13.2 LDM/STM operations.

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Table 3.1 The mapping between the stack and block copy views of the load and store multiple instructions

Example

- PRE
  - r0=0x00080010 r1=0x00000000
  - r2=0x00000000 r3=0x00000000
  - mem32[0x80018]=0x03 mem32[0x80014]=0x02
  - mem32[0x80010]=0x01
- LDMIA r0!, {r1-r3}
- POST
  - r0=0x0008001c r1=0x00000001
  - r2=0x00000002 r3=0x00000003

Example

- PRE
  - r0=0x00009000 r1=0x00000009
  - r2=0x00000008 r3=0x00000007
- STMIB r0!, {r1-r3}
- MOV r1, #1
- MOV r2, #2
- MOV r3,#3
- PRE
  - r0=0x0000900c r1=0x00000001
  - r2=0x00000002 r3=0x00000003
- LDMDA r0!, {r1-r3}
- POST
  - r0=0x00009000 r1=0x00000009
  - r2=0x00000008 r3=0x00000007

3.3 Control flow instructions

- Branch instructions
  - The processor normally executes instructions sequentially, but when it reaches the branch instruction it proceeds directly to the instruction at LABEL instead of executing the instruction immediately after the branch

```
            B       LABEL
            . .
LABEL       . .
```

- Conditional branches
  - The mechanism used to control loop exit is conditional branching

```
        MOV     r0, #0          ; initialize counter
LOOP    ..
        ADD     r0, r0, #1      ; increment loop counter
        CMP     r0, #10         ; compare with limit
        BNE     LOOP            ; repeat if not equal
        ..                      ; else fall through
```

Branch	Interpretation	Normal uses	
B	Unconditional	Always take this branch	
BAL	Always	Always take this branch	
BEQ	Equal	Comparison equal or zero result	Z=1
BNE	Not equal	Comparison not equal or non-zero result	Z=0
BPL	Plus	Result positive or zero	
BMI	Minus	Result minus or negative	
BCC	Carry clear	Arithmetic operation did not give carry-out	
BLO	Lower	Unsigned comparison gave lower	
BCS	Carry set	Arithmetic operation gave carry-out	
BHS	Higher or same	Unsigned comparison gave higher or same	
BVC	Overflow clear	Signed integer operation; no overflow occurred	
BVS	Overflow set	Signed integer operation; overflow occurred	
BGT	Greater than	Signed integer comparison gave greater than	
BGE	Greater or equal	Signed integer comparison gave greater or equal	
BLT	Less than	Signed integer comparison gave less than	
BLE	Less or equal	Signed integer comparison gave less than or equal	
BHI	Higher	Unsigned comparison gave higher	
BLS	Lower or same	Unsigned comparison gave lower or same	

Table 3.2 Branch conditions

3.3 Control flow instructions

- Conditional execution
  - An unusual feature of the ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions

```
        CMP     r0, #5
        BEQ     BYPASS              ; if (r0 != 5) {
        ADD     r1, r1, r0          ;   r1 := r1 + r0 - r2
        SUB     r1, r1, r2          ; }
BYPASS  ..
```

```
CMP     r0, #5              ; if (r0 != 5) {
ADDNE   r1, r1, r0          ;   r1 := r1 + r0 - r2
SUBNE   r1, r1, r2          ; }
..
```

3.3 Control flow instructions

- Conditional execution
  - Conditional execution is invoked by adding the 2-letter condition after the 3-letter opcode
  - It is sometimes possible to write very compact code by cunning use of conditionals

```
; if ((a==b) && (c==d)) e++;
        r0  r1      r2  r3   r4

CMP     r0, r1
CMPEQ   r2, r3
ADDEQ   r4, r4, #1
```

3.3 Control flow instructions

- Branch and link instructions
  - ARM offers this functionality through the branch and link instruction which, as well as performing a branch in exactly the same way as the branch instruction, also saves the address of the instruction following the branch in the link register, r14:

```
        BL      SUBR            ; branch to SUBR
        ..                      ; return to here
SUBR    ..                      ; subroutine entry point
        MOV     pc, r14         ; return
```

  - The subroutine should not call a further, nested, subroutine without first saving r14, otherwise the new return address will overwrite the old one

example

Address		instruction	
100		BL SUB1	*r15=100,r14=104*
104		ADD r1, r2, r3	
		⋮	
400	SUB1	MOV r2, 43	*r15=400,r14=104*
404			
		⋮	
400		MOV r15, lr	*r15=104,r14=104*

Address		instruction	
100		BL SUB1	*r15=100,r14=104*
104		ADD r1, r2, r3	
		⋮	
400	SUB1	MOV r2, 43	*r15=400,r14=104*
404			
		⋮	
440		BL SUB2	*r15=440,r14=444*
		⋮	
480		MOV r15, lr	
		⋮	
820	SUB2	BIC r2, r5, r9	*r15=820,r14=444*
824			
		⋮	
980		MOV r15, lr	*r15=444,r14=444*

3.3 Control flow instructions

- Branch and link instructions (Cont.)
  - The normal mechanism used here is to push r14 onto a stack in memory

```
        BL      SUB1
        ..
SUB1    STMFD   r13!, {r0-r2,r14}  ; save work & link regs
        BL      SUB2
        ..
SUB2    ..
```

```
                AREA    Test, CODE, READONLY
SRAM_BASE       EQU     0x20000200
                ENTRY
                ; set up environment
                LDR     sp, =SRAM_BASE  ;r13 = ptr to stack memory
                ;
                ; your main code is here
                ;
                ; call your routine with a branch and link instruction
                BL      Myroutine
                ;
Myroutine       ; Routine code goes here. First, create space in the
register
                ; file by saving r4-r7, then save the Link Register for
                the return,
                ; all with a single store multiple to the stack
                STMDB   sp!, {r4-r7,lr} ;Save some working registers
                ;
                ; Routine code
                ;
                ; Restore saved registers and move the Link Register
                contents
                ; into the Program Counter, again with one instruction
                LDMIA   sp!, {r4-r7,pc}  ;restore registers and return
                END
```

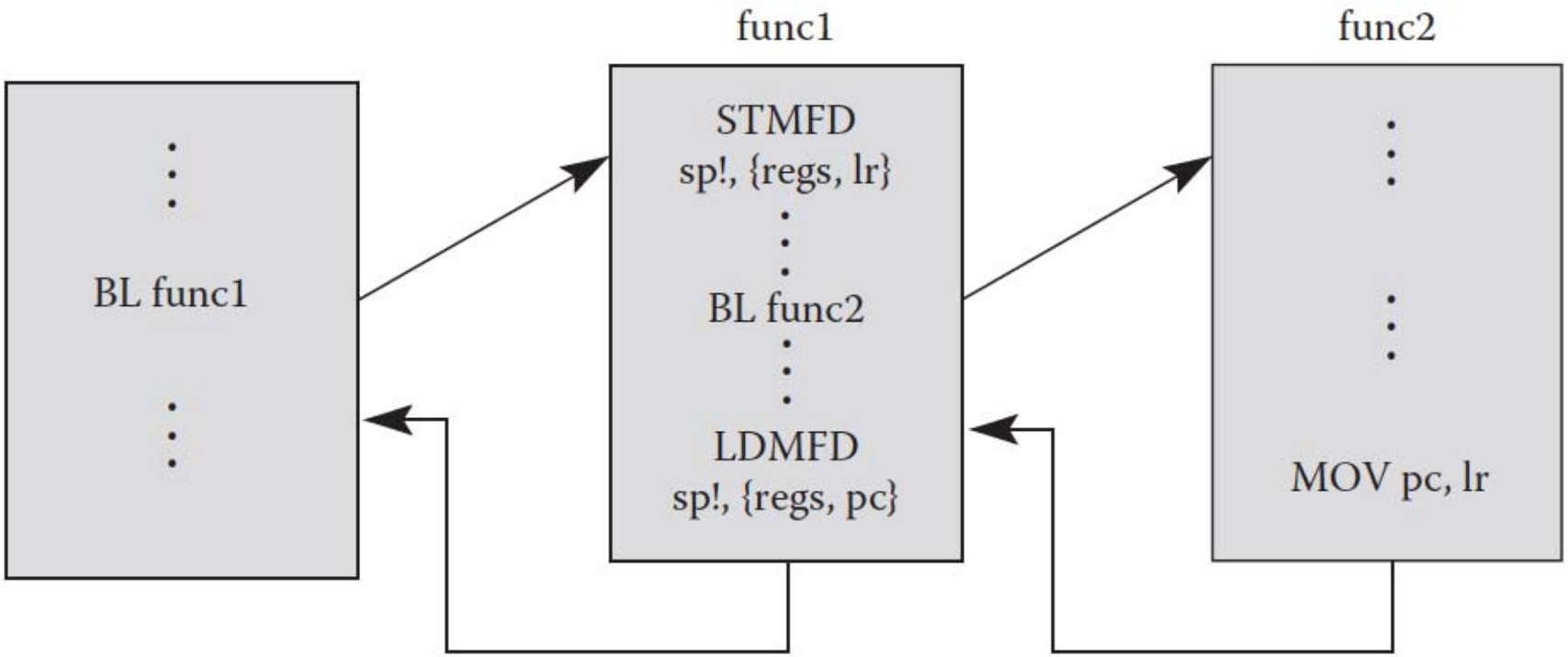


FIGURE 13.5 Stacking the Link Register during entry to a subroutine.

The Program Counter (R15)

- When the processor is executing in ARM state:
  - All instructions are 32 bits in length
  - All instructions must be word aligned
  - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).
- R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link operations are performed,
  calculated from the PC.
- Thus to return from a linked branch
  - `MOV r15,r14`

  or
  - `MOV pc,lr`

3.3 Control flow instructions

- Subroutine return instructions
  - Where the return address has been pushed onto a stack, it can be restored along with any saved work registers using a load multiple instruction:

```
SUB1    STMFD   r13!, {r0-r2,r14}; save work regs & link
        BL      SUB2
        ..
        LDMFD   r13!, {r0-r2,pc} ; restore work regs & return
```

  - The same stack model is used for both the store and the load, ensuring that the correct values will be collected

3.3 Control flow instructions

- Supervisor calls
  - The supervisor is a program which operates at a privileged level, which means that it can do things that a user-level program cannot do directly
  - The supervisor provides trusted ways to access system resources which appear to the user-level program rather like special subroutine accesses
  - The instruction set includes a special instruction, SWI (SoftWare Interrupt), to call these functions (Section 5.6)
  - For example:

```
SWI     SWI_WriteC      ; output  r0[7 : 0]
SWI     SWI_Exit        ; return to monitor
```

SWI Exception

- The sample code of SWI handler
  - STMFD sp!, {r0-r12,r14}
  - LDR r10, [r14,#-4]
  - BIC r10, r10, #0xff000000
  - MOV r1,r13
  - MRS r2, spsr
  - STMFD r13!,{r2}
  - BL swi_jumptable
- The sample code of swi_jumptable
  - swi_jumptable
    - MOV r0, r10
    - B eventsSWIHandler

3.3 Control flow instructions

- Jump tables
  - A programmer sometimes wants to call one of a set of subroutines, the choice depending on a value computed by the program
    - "DCD" directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the left.

```
          BL      JUMPTAB
          ..
JUMPTAB   CMP   r0, #0
          BEQ   SUB0
          CMP   r0, #1
          BEQ   SUB1
          CMP   r0, #2
          BEQ   SUB2
          ..
```

```
          BL      JUMPTAB
          ..
JUMPTAB   ADR     r1, SUBTAB
          CMP     r0, #SUBMAX
          LDRLS   pc, [r1, r0, LSL #2]
          B       ERROR
SUBTAB    DCD     SUB0
          DCD     SUB1
          DCD     SUB2
          ..
```

3.3 Control flow instructions

- Jump tables
  - A programmer sometimes wants to call one of a set of subroutines, the choice depending on a value computed by the program
    - "DCD" directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the left.

```
            BL      JUMPTAB
            ..
JUMPTAB     CMP     r0, #0
            BEQ     SUB0
            CMP     r0, #1
            BEQ     SUB1
            CMP     r0, #2
            BEQ     SUB2
            ..
```

```
            BL      JUMPTAB
            ..
JUMPTAB     ADR     r1, SUBTAB
            CMP     r0, #SUBMAX
            [          ]
            B       ERROR
SUBTAB      DCD     SUB0
            DCD     SUB1
            DCD     SUB2
            ..
```

3.4 Writing simple assembly language program

```
        AREA    HelloW,CODE,READONLY ; declare code area
SWI_WriteC      EQU     &0           ; output character in r0
SWI_Exit        EQU     &11          ; finish program
        ENTRY                        ; code entry point
START   ADR     r1, TEXT             ; r1 -> "Hello World"
LOOP    LDRB    r0, [r1], #1         ; get the next byte
        CMP     r0, #0               ; check for text end
        SWINE   SWI_WriteC           ; if not end print ..
        BNE     LOOP                 ;  .. and loop back
        SWI     SWI_Exit             ; end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                          ; end of program source
```

```
        AREA    BlkCpy,CODE,READONLY
SWI_WriteC      EQU     &0              ; output character in r0
SWI_Exit        EQU     &11             ; finish program
        ENTRY                           ; code entry point
        ADR     r1, TABLE1              ; r1 -> TABLE1
        ADR     r2, TABLE2              ; r2 -> TABLE2
        ADR     r3, T1END               ; r3 -> T1END
LOOP1   LDR     r0, [r1], #4            ; get TABLE1 1st word
        STR     r0, [r2], #4            ; copy into TABLE2
        CMP     r1, r3                  ; finished?
        BLT     LOOP1                   ; if not, do more
        ADR     r1, TABLE2              ; r1 -> TABLE2
LOOP2   LDRB    r0, [r1], #1            ; get next byte
        CMP     r0, #0                  ; check for text end
        SWINE   SWI_WriteC              ; if not end, print ..
        BNE     LOOP2                   ;  .. and loop back
        SWI     SWI_Exit                ; finish
TABLE1  =       "This is the right string!", &0a, &0d, 0
T1END
        ALIGN                           ; ensure word alignment
TABLE2  =       "This is the wrong string!", 0
        END
```

```
        AREA    Hex_Out,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
        LDR     r1, VALUE           ; get value to print
        BL      HexOut              ; call hexadecimal output
        SWI     SWI_Exit            ; finish
VALUE   DCD     &12345678           ; test value
HexOut  MOV     r2, #8              ; nibble count = 8
LOOP    MOV     r0, r1, LSR #28     ; get top nibble
        CMP     r0, #9              ; 0-9 or A-F?
        ADDGT   r0, r0, #"A"-10     ; ASCII alphabetic
        ADDLE   r0, r0, #"0"        ; ASCII numeric
        SWI     SWI_WriteC          ; print character
        MOV     r1, r1, LSL #4      ; shift left one nibble
        SUBS    r2, r2, #1          ; decrement nibble count
        BNE     LOOP                ; if more do next nibble
        MOV     pc, r14             ; return
        END
```